Francisco Zamora Treviño A01570484

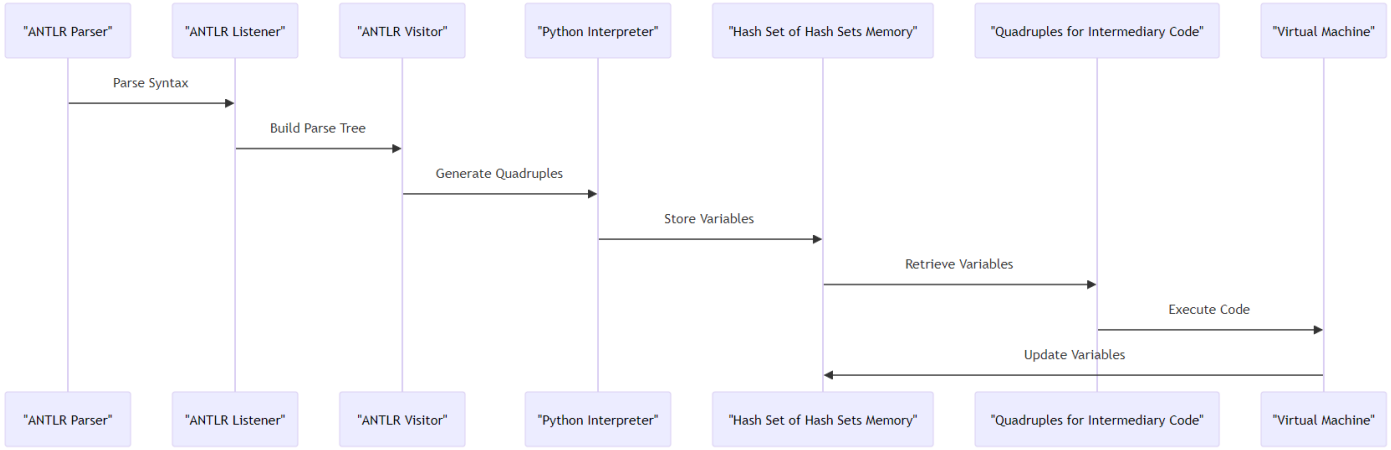
BabyDuck Programming Language Docs

Overview

This implementation of the programming language has the following features:

1. **Semantic/Intermediary generation** layer is based on the **visitor** pattern, approximating functional programming inside of the Python language.
2. **Preprocessing** of Intermediary code to leverage host language's object representation.
3. Emphasis con Virtual Machine code **simplicity**.

Sequence Diagram



Files Overview

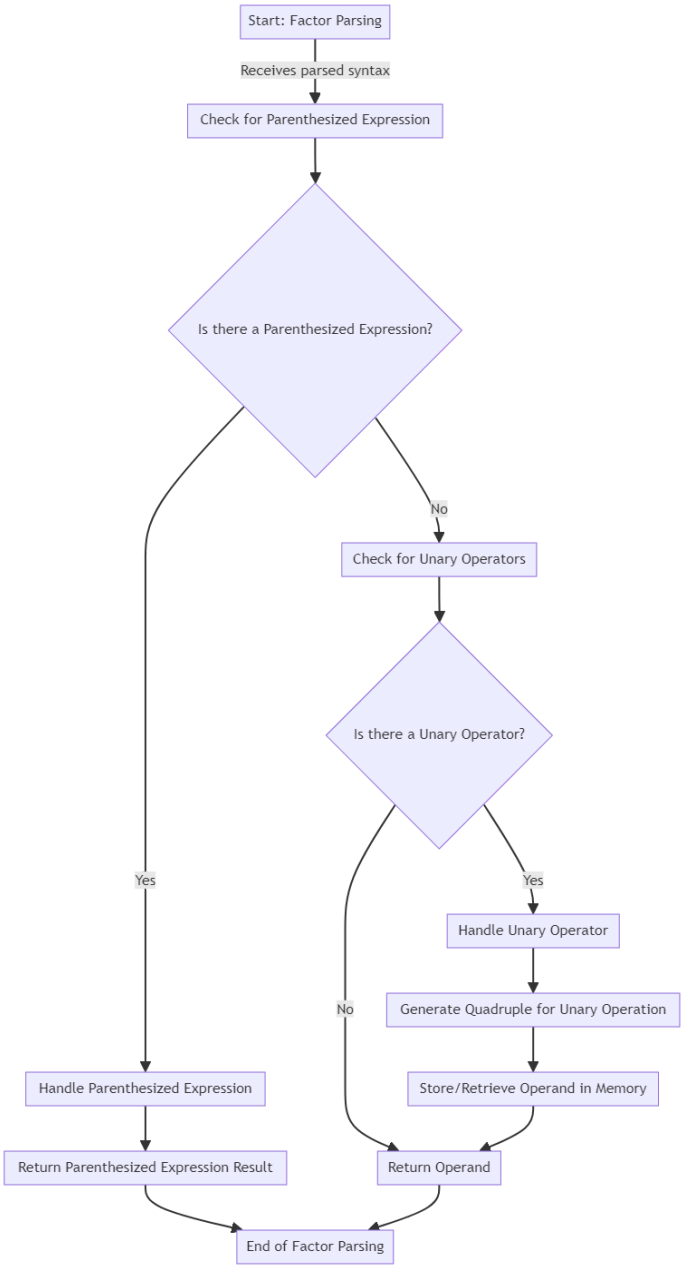
- **`README.md`**: General information about the project.
- **`Expr.g4`, `baby_duck_grammar.g4`**: ANTLR grammar files for the language.
- **`driver.py`**: The main driver script for the interpreter.
- **`Quadruple.py`**: Defines the quadruple structure used in the language.
- **`SemanticCube.py`**: Manages semantic rules and operations.
- **`mem_tables.py`**: Implements the memory structure.
- **`vm.py`**: The virtual machine for executing the code.

Visitor

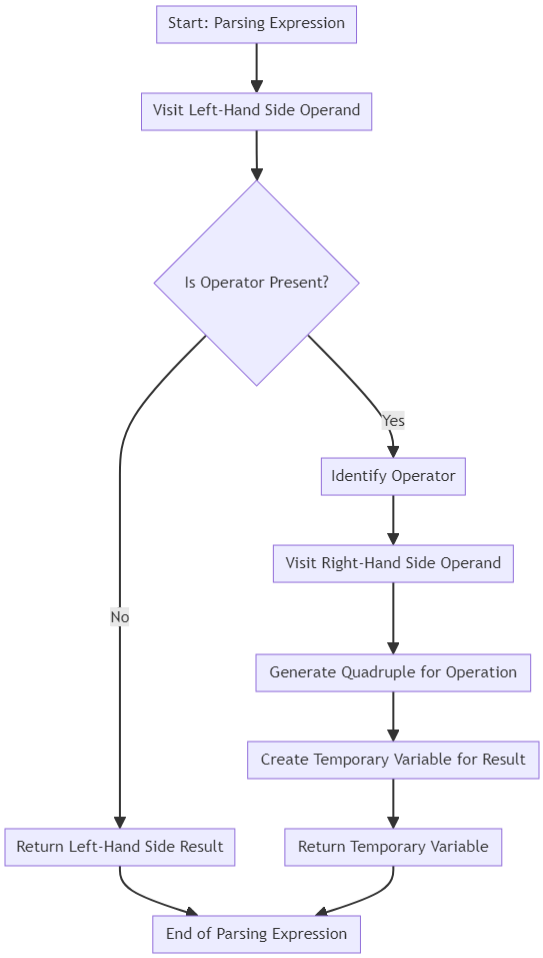
The visitor pattern was preferred over other methods primarily because it allows for the separation of concerns of each of the token's intermediary code translation methods. Besides, it allows for a more "code-like" experience of building the translation routines.

As it stems from the functional programming style, it carries the benefits of being less bug-prone, as each of the components is separated and the stacks typically required by algorithms like Shunting-Yard are implicit structures instead of explicit objects.

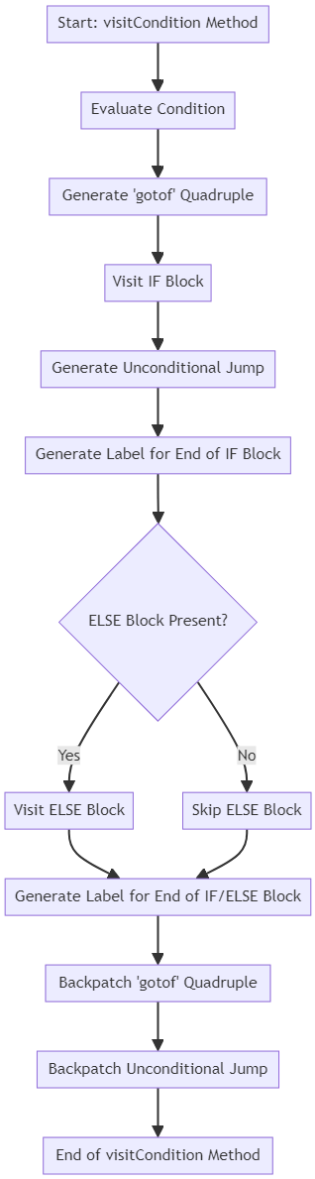
1. **`visitFactor`**:



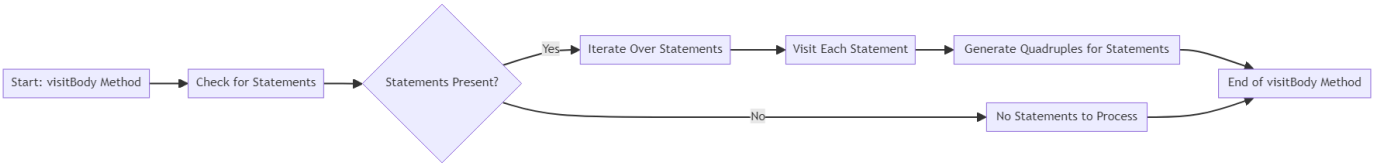
2. **`visitExpression`**:



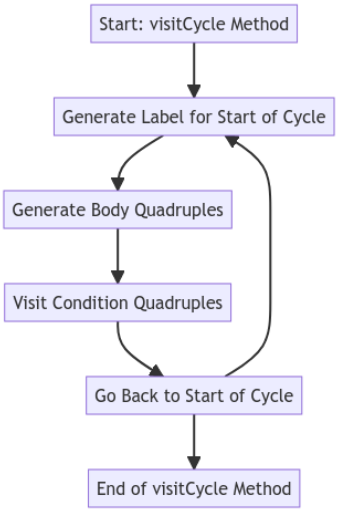
3. **`visitCondition`**:



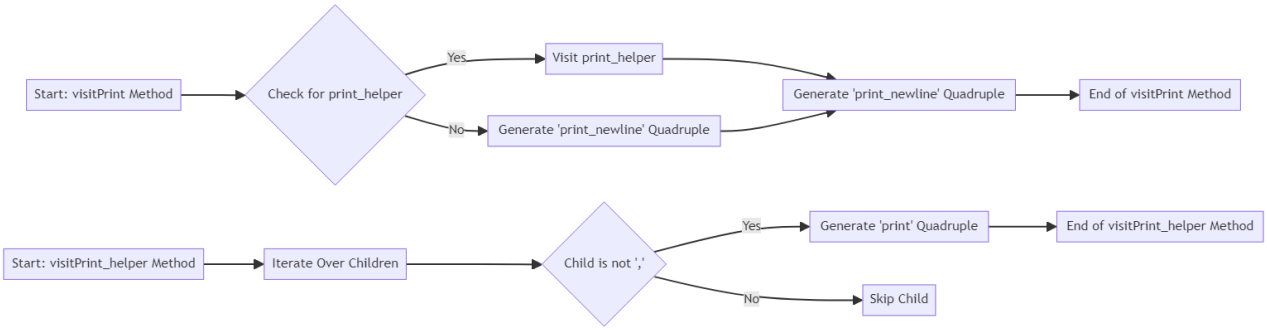
4. **`visitBody`**:



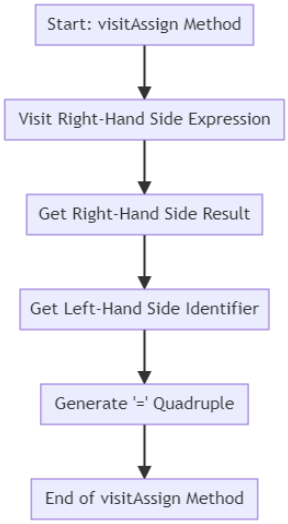
5. **`visitCycle`**:



6. **`visitPrint`** and **`visitPrint_helper`**:



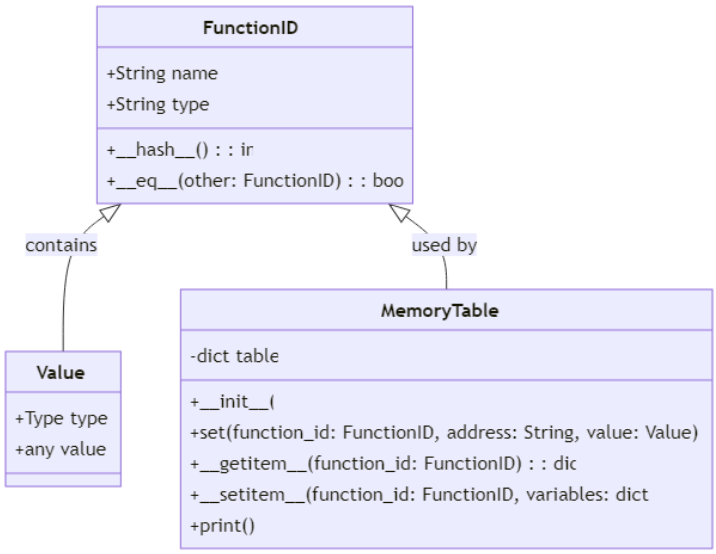
7. **`visitAssign`**:



Memory management

The memory model is essentially a hash set of hash sets. Each function in the program has its own set of variables, identified by the **FunctionID**. This structure allows for efficient storage and retrieval of variables and their values, scoped to specific functions.

The variables are stored using a wrapper function called **Value**, which holds a tag that includes an **Enumerator** that determines the type of the actual data being held. This is done to increase performance as it avoids parsing the values at runtime, which given Python's high level architecture, would require the value to be tested against the (constant) number of types that the BabyDuck schema contemplates.



Algorithms

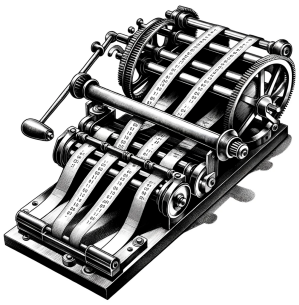
1. **Depth-First Search:** The Visitor employs a DFS algorithm to traverse the parse tree. This means it explores as far as possible along each branch before backtracking.

2. **Quadruple Generation:** In many of these visit methods, quadruples (intermediate code representations) are generated. This step is crucial for the subsequent execution of the code by the virtual machine. Quadruples are a form of intermediate representation that makes it easier to execute or translate into another language.

3. **Recursive Traversal:** For nodes that contain other nodes (like a body of statements), the Visitor recursively visits each child node. This recursive nature ensures that every part of the parse tree is explored and processed.

4. **Handling Special Cases:** The Visitor also handles special cases, such as unary operators in expressions, or the presence of else blocks in conditional statements. This is done by checking specific conditions in the parse tree and executing the appropriate logic.

5. **Backpatching:** In certain constructs like conditional statements and loops, backpatching is used. This involves revisiting previously generated quadruples to fill in missing information, such as jump targets, once they are known.



Operators and Virtual Machine

The virtual machine simply operates on a Program Counter that points to an array of quadruples. The only I/O that the machine has (besides the memory) is through the PRINT operator, which invokes the host language's implementation for console printout.

1. **PLUS (`Operator.PLUS`)**
    - **Left Operand**: First value for addition.
    - **Right Operand**: Second value for addition.
    - **Result**: Memory address where the sum is stored.
2. **ASSIGN (`Operator.ASSIGN`)**
    - **Left Operand**: Source value.
    - **Right Operand**: Not used.
    - **Result**: Memory address where the left operand value is assigned.
3. **PRINT (`Operator.PRINT`)**
    - **Left Operand**: Value to be printed.
    - **Right Operand**: Not used.
    - **Result**: Not used. Directly prints the left operand value.
4. **PRINT_NEWLINE (`Operator.PRINT_NEWLINE`)**
    - **Left Operand**: Not used.
    - **Right Operand**: Not used.
    - **Result**: Not used. Triggers printing of a newline.
5. **MINUS (`Operator.MINUS`)**
    - **Left Operand**: First value for subtraction.
    - **Right Operand**: Second value for subtraction.
    - **Result**: Memory address where the difference is stored.
6. **TIMES (`Operator.TIMES`)**
    - **Left Operand**: First value for multiplication.
    - **Right Operand**: Second value for multiplication.
    - **Result**: Memory address where the product is stored.
7. **DIVIDE (`Operator.DIVIDE`)**
    - **Left Operand**: Dividend.
    - **Right Operand**: Divisor.
    - **Result**: Memory address where the quotient is stored.
8. **LESS_THAN (`Operator.LESS_THAN`)**
    - **Left Operand**: First value for comparison.
    - **Right Operand**: Second value for comparison.
    - **Result**: Memory address where the boolean result of the comparison is stored.
9. **GREATER_THAN (`Operator.GREATER_THAN`)**
    - **Left Operand**: First value for comparison.
    - **Right Operand**: Second value for comparison.
    - **Result**: Memory address where the boolean result of the comparison is stored.
10. **GOTOT (`Operator.GOTOT`)**
    - **Left Operand**: Condition value.
    - **Right Operand**: Not used.
    - **Result**: Memory address or label to jump to if the condition is true.
11. **GOTOF (`Operator.GOTOF`)**
    - **Left Operand**: Condition value.
    - **Right Operand**: Not used.
    - **Result**: Memory address or label to jump to if the condition is false.
12. **GOTO (`Operator.GOTO`)**
    - **Left Operand**: Not used.
    - **Right Operand**: Not used.
    - **Result**: Memory address or label for unconditional jump.
13. **EQUAL (`Operator.EQUAL`)**

- **Left Operand**: First value for equality check.
- **Right Operand**: Second value for equality check.
- **Result**: Memory address where the boolean result of the equality check is stored.

Grammar

```
grammar baby_duck_grammar;
// Francisco Zamora Treviño A01570484
ID: [a-zA-Z] ([a-zA-Z] | [0-9])*;
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
FLOAT: [0-9]+ '.' [0-9]+;
string: '"'*?'"';

program: 'program' ID ';' vars? program_post_var;

program_post_var: funcs* 'main' body 'end';

body: '{' statement* '}';

statement: assign | condition | cycle | f_call | print;

type: 'int' | 'float' | 'bool' | 'void';

assign: ID '=' expression ';';

expression: exp (rel_op expression)?;

rel_op: '<' | '>' | '!=' | '==' | '<=' | '>=c';

cte: INT | FLOAT;

print: 'print' '(' print_helper ')' ';';

print_helper: (expression | string) (',' (expression | string))*;

f_param_list: (f_param_list_helper (',' f_param_list_helper)*)?;

f_param_list_helper: (ID ':' type);

funcs: function_id '(' f_param_list ')' '[' vars? body ']' ';';

function_id: type ID;

vars: 'var' vars_declarations+;
```

```
vars_declarations: (ID (',' ID)* ':' type ';');

cycle:   'do' body 'while' '(' expression ')' ';';

while_keyword: 'while';

condition: 'if' '(' expression ')' body condition_else? ';';

condition_else: 'else' body;

operator: ('+' | '-');

exp: term (operator exp)?;

term: factor (term_operator term)?;

term_operator: ('*' | '/');

factor: parenthesized_expression | (factor_operator? (ID | cte));

factor_operator: ('+' | '-');

parenthesized_expression: '(' expression ')';

f_call: ID '(' f_call_helper? ')' ';';

f_call_helper: expression (',' expression)*;
```